Apunte 24 — Mantenimiento

Centralización de Logs y Monitoreo de Aplicaciones

Introducción

En el apunte anterior aprendimos a manejar excepciones y registrar logs localmente.
Sin embargo, en un entorno distribuido con múltiples microservicios, mantener los logs de manera aislada puede dificultar el diagnóstico y la trazabilidad.

La **centralización de logs** permite agrupar los registros de distintas instancias o servicios en un único punto de consulta, facilitando el monitoreo, la auditoría y el análisis del comportamiento general del sistema.

Logs en arquitecturas de Microservicios

Ya hemos conversado los beneficios que trae la arquitectura de microservicios: Tolerancia a fallas, redundancia, independencia entre servicios, mantenibilidad, escalabilidad, etc. Sin embargo, no todo es tan fácil, porque una arquitectura de este tipo trae una mayor complejidad a la hora de realizar la administración de la infraestrcutura y el mantenimiento; y uno de los elementos que se vuelven más complicados son los logs.

En general, siempre que surge un problema en producción en un Sistema/Software y necesitamos saber qué pasó, vamos a ver los logs, donde probablemente podamos encontrar la causa del problema. Los Logs son el registro de todas las salidas del servidor, dependiendo la instancia en la que estemos estas salidas podrán ir a la pantalla, a un archivo, al log del contenedor de despliegue o incluso a una base de datos.

Y además, como vimos, estos logs se categorizan en: Información general de trazabilidad (INFO), Información de Depuración (DEBUG), o Situaciones Críticas (ERROR) entre las demás categorías que ya documentamos en el apunte anterior pero podemos mencionar estas para resumir el concepto.

¿Cuál es el problema?

Cada una de las APIs de una arquitectura de microservicios en Java corren sobre la JVM propia. Es decir tenemos varias JVM dando vida al backend de nuestro Sistema, y cada una de las JVM en general el loguea usando una librería que suele ser Slf4j, Logback o algo similar. En todo el ecosistema Java, el logueo está bastante estandarizado con estas librerías. Estas librerías en general pueden escribir el log en la terminal o guardarlo en un archivo de texto de forma que luego podamos revisar el archivo de texto existente, esto último lo denominamos loguear con persistencia. Y si a esa arquitectura le agregamos docker, además, cada jvm ejecuta en un contenedor independiente.

Como dijimos, tenemos múltiples aplicaciones ejecutando, cada una con sus propios logs, podemos vivir esto al ejecutar en desarrollo y tener la terminal de cada API por separado, el problema que surge es

evidente, en un entorno con un poco más de realidad y desatendido es imposible tener que andar revisando diferentes terminales para encontrar el log de una situación específica y por ello la pregunta:

¿Dónde guardamos los logs?

Esto ya representa un problema en sí. La forma más básica de loguear con persistencia implica guardar en un archivo dentro de la máquina donde corre el programa todo el log que genera la aplicación, aumentando el tiempo de procesamiento, porque escribir a un archivo en disco suele ser un proceso costoso. Ya vimos en el apunte anterior como almacenar esos logs en archivos e incluso y creando nuevos archivos a partir de ciertas características.

Esta es una solución un poco rústica porque digamos que tenemos un problema en el servicio X y queremos leer los logs para entender qué es lo que sucedió y encontrar la solución. Tenemos N apis/servicios dentro de la aplicación, por lo que deberíamos entrar a cada archivo de log de estas apis/servicios y buscar en CADA UNO por separado la información que queremos.

Leer logs de una sola API ya es un proceso costoso, pero si el problema requiere revisar los logs de varias APIs, sería notablemente más costoso. Claramente esta solución no escala y demanda muchísimo tiempo. Y si a esto le agregamos la redundancia de cada uno de los microservicios el problema se vuelve exponencial.

Además, tenemos otro inconveniente: si tenemos un problema en alguna instancia y, por ejemplo, el host tiene un problema de hardware, todo el log que se encontraba en esa instancia se perdería para siempre, junto con, quizás, otra valiosa información. Recuerden, tenemos redundancia de servicios para poder escalar horizontalmente, pero también para tener tolerancia a fallos. Y estas cosas parece que no suceden pero sí, ocurren, y bastante más seguido de lo que pensamos.

Entonces podemos concluir que tener los logs dentro de cada instancia es algo que no nos sirve, pero, ese es un problema que veremos más adelante por ahora nos vamos a concentrar en el comienzo que es lograr generar logs útiles y accesibles.

Logs en entornos Docker

Introducción a logs en Docker

Cuando desplegamos nuestras aplicaciones Java dentro de contenedores Docker, el manejo de logs adquiere un papel central en la observabilidad del sistema. Cada contenedor genera su propia salida de logs, y entender **dónde se guardan, cómo acceder a ellos y cómo centralizarlos** es fundamental para el diagnóstico y monitoreo.

En este apartado ampliamos los conceptos básicos e incluimos ejemplos prácticos de uso, configuraciones de `docker` y `docker compose`, así como las salidas esperadas y su análisis.

Dónde se guardan los logs de los contenedores Docker

Cada contenedor Docker mantiene sus propios logs, los cuales se redirigen por defecto a los flujos estándar `stdout` (salida estándar) y `stderr` (salida de errores). Docker almacena estos registros en el host, generalmente en:

```
/var/lib/docker/containers/<container_id>/<container_id>-json.log
```

Cada archivo de log se genera en formato **JSON**, con cada línea representando un evento. Ejemplo de contenido:

```
{"log":"2025-03-11T10:32:04.123Z INFO  [main] utnfc.isi.back.App - Aplicación iniciada\n","stream":"stdout","time":"2025-03-11T10:32:04.124Z"}
```

> ⚠ En general no recomendamos leer directamente estos archivos, sino usar los comandos de Docker o herramientas externas para su lectura y análisis.

Visualización de logs con comandos Docker

🟢 Ver los logs de un contenedor específico

```
docker logs app-service
```

Salida esperada:

```
2025-03-11 10:32:04 INFO  utnfc.isi.back.App : Aplicación iniciada
2025-03-11 10:32:05 INFO  utnfc.isi.back.controller.HomeController : Servidor escuchando en puerto 8080
```

🟢 Seguir logs en tiempo real

```
docker logs -f app-service
```

> La opción `-f` (follow) nos permite observar los logs en vivo, como el comando `tail -f` en Linux.

🟢 Filtrar logs recientes

```
docker logs --since 10m app-service
```

Mostramos solo los registros generados en los últimos 10 minutos.

🟢 Mostrar solo errores (`stderr`)

```
docker logs --stderr app-service
```

Ideal para revisar rápidamente fallas sin ruido de logs informativos.

Logs en entornos con múltiples contenedores

Cuando tenemos varias aplicaciones ejecutándose en el mismo servidor (por ejemplo, microservicios), los logs de cada contenedor se gestionan de forma independiente. Para obtener una vista consolidada, podemos usar Docker Compose.

🔹 **Ver todos los logs de un entorno Compose**

```
docker compose logs
```

Esto muestra los logs de todos los servicios definidos en el archivo `docker-compose.yml`, ordenados cronológicamente. Ejemplo:

```
app-gateway-1     | 2025-03-11 10:45:10 INFO  Gateway iniciado en puerto
8081
personas-service-1| 2025-03-11 10:45:11 INFO  Servicio de Personas
escuchando en 8082
tramites-service-1| 2025-03-11 10:45:12 INFO  Servicio de Trámites activo
en 8083
```

🔹 **Filtrar por un servicio específico**

```
docker compose logs personas-service
```

Mostramos solo los logs correspondientes a `personas-service`.

🔹 **Seguir logs en tiempo real de todos los servicios**

```
docker compose logs -f
```

Podemos observar la interacción en vivo entre los servicios, lo cual resulta muy útil para depurar flujos distribuidos.

> 💡 **Consejo:** usar prefijos en los logs de nuestras aplicaciones (por ejemplo, incluir el nombre del servicio en el mensaje) mejora la interpretación de los logs combinados.

Configuración del driver de logs

Docker usa diferentes **drivers de logging** para definir cómo se almacenan y gestionan los registros. El más común es `json-file`, que guarda los logs en formato JSON.

Ejemplo de configuración en `docker-compose.yml`:

```
services:
  app-service:
    build: .
    container_name: app-service
    ports:
      - "8080:8080"
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

Esta configuración limita el tamaño máximo de los archivos de log a **10 MB** y conserva un historial de los **3 archivos más recientes** (rotación automática).

Otros drivers disponibles incluyen:

- `syslog`: envía logs a un servidor syslog externo.
- `journald`: integra los logs con `systemd`.
- `fluentd`: envía logs a un servicio Fluentd para centralización.
- `awslogs`, `gelf`, `splunk`: para entornos cloud o herramientas empresariales.

Analizando logs con Compose y mejorando la legibilidad

El comando `docker compose logs` puede producir una salida extensa. Para mejorar el análisis, podemos aplicar técnicas como:

🔹 **Filtrar por palabras clave**

```
docker compose logs | grep ERROR
```

🔹 **Guardar los logs en un archivo para análisis posterior**

```
docker compose logs > logs_sistema.txt
```

🔹 **Mostrar solo las últimas líneas**

```
docker compose logs --tail=50
```

🔹 **Combinar opciones**

```
docker compose logs -f --tail=100 personas-service
```

Esto nos permite seguir en vivo las últimas 100 líneas del servicio `personas-service`.

Buenas prácticas para aplicaciones Java en Docker

- Configurar nuestro logger (Logback o Log4j2) para escribir en **stdout** y **stderr**, no en archivos internos del contenedor.
- Prefijar cada log con el nombre del servicio o microservicio (`spring.application.name` en `application.yml`).
- Evitar logs excesivos en `DEBUG` o `TRACE` en entornos productivos.
- Mantener los logs estructurados y legibles (JSON o texto claro).
- Implementar una convención común de formato entre microservicios.

Ejemplo de configuración en `application.yml`:

```
spring:
  application:
    name: personas-service
logging:
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
[%X{traceId}] %msg%n"
```

El campo `%X{traceId}` nos permite incluir un identificador único por solicitud (si usamos Spring Cloud Sleuth u otra librería de trazas), facilitando el seguimiento entre servicios.

Resumen

La correcta configuración y análisis de logs en entornos Docker es esencial para lograr visibilidad y diagnósticos precisos. Aprovechar comandos como `docker compose logs`, configurar los drivers de logging y mantener una estructura uniforme en los mensajes nos permite simplificar la tarea de monitoreo y preparar el terreno para una futura **centralización de logs** mediante herramientas como Loki, ELK o Promtail.

Trazabilidad de sesión en arquitecturas de microservicios

Introducción al problema

En una arquitectura de microservicios, una misma solicitud de nuestros usuarios puede atravesar **múltiples servicios** antes de devolver una respuesta. Cada servicio genera sus propios logs de manera independiente, lo que hace extremadamente complejo reconstruir el flujo completo de una transacción o sesión cuando ocurre un problema.

Por ejemplo, una petición que se origina en nuestro `gateway` puede pasar por un `servicio de autenticación`, un `servicio de usuarios`, un `servicio de pagos` y finalmente un `servicio de`

`notificaciones`. Si cada uno genera sus logs por separado sin un identificador común, nos resulta prácticamente imposible correlacionarlos para entender el recorrido de esa solicitud.

La pérdida de contexto entre microservicios

Cada servicio maneja su propio contexto de ejecución, sus hilos y su propio logger. Cuando hacemos llamadas HTTP entre servicios, el contexto se pierde a menos que **propaguemos un identificador único de trazabilidad**. Esto genera síntomas comunes:

- Logs desconectados: cada microservicio registra eventos sin relación visible.
- Diagnósticos ineficientes: no sabemos cuál fue el orden real de las llamadas.
- Dificultad para reproducir errores específicos de un usuario o sesión.

Ejemplo clásico del problema:

```
[Gateway]  INFO  -> Procesando solicitud POST /api/usuarios/login
[AuthService]  INFO  -> Validando credenciales del usuario
[UserService]  INFO  -> Consultando datos del usuario
[PaymentService]  ERROR -> Timeout al procesar pago
```

Sin un identificador común, no podemos saber si esas líneas pertenecen a la misma transacción o a distintas solicitudes concurrentes.

La solución: identificadores de correlación

Para resolver este problema, cada solicitud que ingresa a nuestro sistema debe portar un **Correlation ID** (identificador de correlación) que se propague a lo largo de todos los microservicios que participan en la transacción.

Podemos generar este ID en el `gateway` (cuando la solicitud llega al sistema) y enviarlo a través de un encabezado HTTP estándar, por ejemplo:

```
X-Correlation-ID: 2b97d1af-2e7a-4a35-a3db-98765fc22e1d
```

Cada servicio debe:

1. Leer el encabezado `X-Correlation-ID` si ya existe, o generarlo si no viene definido.
2. Incluir ese valor en todos sus logs.
3. Propagarlo cuando invoque a otros microservicios.

De este modo, todos los logs de una misma sesión compartirán el mismo identificador.

Ejemplo del resultado esperado:

```
[Gateway] [CID=2b97d1af] Procesamos solicitud POST /api/usuarios/login
[AuthService] [CID=2b97d1af] Validamos credenciales del usuario
```

```
[UserService] [CID=2b97d1af] Consultamos datos del usuario
[PaymentService] [CID=2b97d1af] Timeout al procesar pago
```

Ahora podemos filtrar por `CID=2b97d1af` y obtener la secuencia completa de la solicitud, aunque haya pasado por diferentes servicios y contenedores.

Implementación en Spring Boot

Podemos lograr esta trazabilidad de forma sencilla usando un filtro (`OncePerRequestFilter`) que capture o cree el `Correlation ID` al inicio de cada request.

Ejemplo:

```
@Component
public class CorrelationIdFilter extends OncePerRequestFilter {

    private static final String HEADER_NAME = "X-Correlation-ID";

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String correlationId = request.getHeader(HEADER_NAME);
        if (correlationId == null || correlationId.isBlank()) {
            correlationId = UUID.randomUUID().toString();
        }

        MDC.put("correlationId", correlationId);
        response.addHeader(HEADER_NAME, correlationId);

        try {
            filterChain.doFilter(request, response);
        } finally {
            MDC.remove("correlationId");
        }
    }
}
```

Luego, en `logback-spring.xml`, agregamos el campo al patrón de log:

```
<pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread] [%X{correlationId}]
%logger{36} - %msg%n</pattern>
```

Esto nos permite que cada línea de log incluya automáticamente el `Correlation ID`.

Propagación del Correlation ID entre microservicios

Para mantener el mismo ID a través de las llamadas REST, incluimos el encabezado HTTP en los clientes que realizan peticiones entre servicios.

Ejemplo usando RestTemplate:

```
@Component
public class CorrelationRestTemplateInterceptor implements
ClientHttpRequestInterceptor {
    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution)
            throws IOException {
        String correlationId = MDC.get("correlationId");
        if (correlationId != null) {
            request.getHeaders().add("X-Correlation-ID", correlationId);
        }
        return execution.execute(request, body);
    }
}
```

Con esta estrategia, el identificador se mantiene a lo largo de toda la cadena de llamadas, permitiéndonos reconstruir el flujo completo de la solicitud.

Análisis del flujo distribuido

Con la trazabilidad activada, podemos observar en los logs un flujo coherente entre servicios:

```
[CID=7a3f91] Gateway → AuthService → UserService → PaymentService
```

Esto nos permite:

- Detectar cuellos de botella o latencias entre servicios.
- Identificar cuál fue el microservicio donde ocurrió la excepción original.
- Correlacionar errores de usuario con trazas exactas del sistema.

> 🧠 En entornos más avanzados, esta misma idea se integra con herramientas de trazabilidad distribuida como **Spring Cloud Sleuth**, **Zipkin** o **Jaeger**, pero el concepto básico de Correlation ID es la base de toda trazabilidad de logs.

En resumen

La trazabilidad de sesión es un desafío clave en arquitecturas de microservicios. Sin una estrategia de correlación, nuestros logs distribuidos pierden valor diagnóstico. Implementar un **Correlation ID** consistente entre servicios nos permite seguir el recorrido completo de una solicitud, identificar la causa raíz de los problemas y fortalecer nuestra capacidad de observación del sistema.

Centralización de logs

De la trazabilidad a la centralización de logs

Una vez que implementamos la trazabilidad mediante `Correlation ID`, logramos identificar el recorrido completo de una solicitud dentro de nuestros microservicios. Sin embargo, todavía enfrentamos un desafío importante: **nuestros logs siguen distribuidos entre múltiples contenedores o servidores**, lo que complica su análisis integral.

Cuando queremos diagnosticar un error que ocurre en una transacción que atraviesa varios microservicios, necesitamos poder buscar y correlacionar los eventos de todos ellos en un solo lugar. Para resolver este problema, implementamos la **centralización de logs**.

La centralización nos permite recopilar, almacenar y visualizar todos los registros del sistema en un repositorio único, facilitando:

- La búsqueda rápida de errores o eventos específicos.
- El análisis del flujo completo de una solicitud distribuida.
- La detección de patrones de error o comportamiento anómalo.
- La creación de paneles de monitoreo y auditoría.

Fundamentos de los stacks de observabilidad

El proceso de centralización de logs se apoya en arquitecturas conocidas como **stacks de observabilidad**, que integran tres etapas principales:

1. **Recolección:** capturamos los logs desde las salidas estándar (`stdout` y `stderr`) o desde los archivos generados por los contenedores Docker.
2. **Almacenamiento:** enviamos los logs a una base de datos optimizada para búsquedas de texto, como **Elasticsearch** o **Loki**.
3. **Visualización:** utilizamos herramientas como **Kibana** o **Grafana** para consultar, filtrar y analizar los registros.

Estos stacks permiten transformar los logs dispersos en información centralizada y navegable, aportando trazabilidad completa y contexto de ejecución.

Los más utilizados son:

- 🟢 **ELK Stack** (Elasticsearch, Logstash, Kibana): una solución madura y potente para análisis centralizado de logs.
- 🔵 **Loki Stack** (Promtail, Loki, Grafana): una alternativa moderna y ligera diseñada por Grafana Labs, ideal para entornos Docker y Kubernetes.

> 📘 En el anexo del apunte incluimos una descripción más detallada de estos stacks, sus componentes y principales ventajas comparativas.

Problemas del enfoque local

- Dificultad para buscar y correlacionar errores entre microservicios.
- Pérdida de logs al eliminar contenedores o reiniciar servicios.
- Falta de visibilidad global del sistema.

En entornos donde cada contenedor escribe sus logs localmente, identificar el origen de un error o el flujo de una solicitud puede ser una tarea lenta y fragmentada. La centralización soluciona esto al consolidar todos los eventos en un único punto de consulta.

Arquitectura general de logs centralizados

El flujo típico de logs en una arquitectura distribuida con observabilidad es el siguiente:

```
Microservicios → stdout/stderr → Recolector de logs (Filebeat / Promtail)
→
→ Almacenamiento (Elasticsearch / Loki) → Visualización (Kibana / Grafana)
```

Cada componente cumple un rol específico:

- **Recolector:** lee los logs de los contenedores y los envía a la base central.
- **Almacenamiento:** indexa los registros, permitiendo búsquedas y filtros.
- **Visualización:** ofrece una interfaz para exploración y análisis en tiempo real.

Ejemplo conceptual con Docker Compose

El siguiente ejemplo muestra una configuración simplificada de entorno Docker Compose que incluye una aplicación, un recolector (Promtail), un backend de logs (Loki) y una herramienta de visualización (Grafana):

```yaml
version: "3.9"
services:
  app1:
    image: myapp1:latest
    logging:
      driver: "json-file"

  app2:
    image: myapp2:latest

  log-collector:
    image: prom/promtail:latest
    volumes:
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - ./config/promtail.yml:/etc/promtail/config.yml
    command: -config.file=/etc/promtail/config.yml

  loki:
    image: grafana/loki:latest
    ports:
      - "3100:3100"

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
```

> 💡 Este ejemplo es solo conceptual. En el anexo final del apunte incluimos una descripción más detallada del **Stack ELK** y del **Stack Loki**, con sus diferencias y configuraciones básicas.

Para concluir sobre la centralización de logs

La centralización de logs representa el siguiente paso natural luego de la trazabilidad: mientras que el `Correlation ID` nos permite seguir una transacción entre servicios, la centralización nos brinda una **visión global** de todo el ecosistema. Esto nos permite detectar fallas más rápido, analizar patrones de error, y consolidar la base de la **observabilidad moderna** en entornos distribuidos.

Monitoreo con Spring Boot Actuator

Qué es Actuator

Spring Boot Actuator agrega endpoints de monitoreo que exponen información sobre el estado y métricas de la aplicación.
Permite verificar el *health check*, métricas de rendimiento y detalles de configuración.

Configuración básica

Agregar la dependencia:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Habilitar los endpoints en `application.yml`:

```
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,loggers
  endpoint:
    health:
      show-details: always
```

Endpoints principales

Endpoint	Descripción
`/actuator/health`	Verifica el estado de la aplicación.
`/actuator/info`	Muestra información personalizada.
`/actuator/metrics`	Expone métricas internas del sistema.

Endpoint	Descripción
`/actuator/loggers`	Permite modificar niveles de log en tiempo de ejecución.

Ejemplo de uso

```
curl http://localhost:8080/actuator/health
curl http://localhost:8080/actuator/metrics/jvm.memory.used
```

Introducción a métricas y observabilidad

Qué es la observabilidad

La **observabilidad** combina tres pilares fundamentales:

- **Logs:** registro de eventos y errores.
- **Métricas:** valores numéricos medibles a lo largo del tiempo.
- **Trazas:** seguimiento de peticiones distribuidas entre servicios.

Métricas personalizadas con Micrometer

Spring Boot utiliza **Micrometer** para recolectar métricas que pueden exportarse a sistemas como Prometheus.
Ejemplo básico:

```
@Component
public class SolicitudesCounter {
    private final Counter contador;

    public SolicitudesCounter(MeterRegistry registry) {
        contador = Counter.builder("solicitudes_total")
                          .description("Número total de solicitudes
procesadas")
                          .register(registry);
    }

    public void incrementar() {
        contador.increment();
    }
}
```

Acceso a métricas desde Actuator

Una vez configurado Actuator, las métricas personalizadas aparecerán automáticamente en `/actuator/metrics`.

🧩 Anexo — Stacks de Observabilidad: ELK vs Loki

Introducción al Anexo

En esta sección ampliamos los conceptos mencionados en el bloque de **Centralización de Logs**, profundizando en los dos stacks más utilizados en entornos modernos: **ELK Stack** y **Loki Stack**. Ambos tienen como objetivo **centralizar y analizar los logs de aplicaciones distribuidas**, pero difieren en su arquitectura, consumo de recursos y enfoque técnico.

ELK Stack — Elasticsearch, Logstash y Kibana

El **ELK Stack** es una solución madura y ampliamente adoptada para la recolección, almacenamiento y análisis de logs. Está compuesto por tres herramientas principales:

Elasticsearch (almacenamiento e indexación)

- Es una **base de datos orientada a documentos**, altamente escalable y optimizada para búsquedas de texto.
- Recibe los logs desde Logstash o Beats y los almacena en índices.
- Permite realizar consultas complejas con filtros, agregaciones y visualizaciones en tiempo real.

📘 *Ejemplo de índice de logs:*

```
{
  "timestamp": "2025-03-11T10:35:42Z",
  "level": "ERROR",
  "service": "usuarios-service",
  "message": "Timeout al obtener datos del usuario",
  "correlationId": "2b97d1af-2e7a-4a35-a3db-98765fc22e1d"
}
```

Logstash (recolección y procesamiento)

- Se encarga de **recibir logs** de múltiples fuentes (archivos, flujos TCP, Beats, Kafka, etc.) y **procesarlos** antes de enviarlos a Elasticsearch.
- Permite aplicar filtros, parsear estructuras JSON, y enriquecer los datos con etiquetas, niveles o metadatos.

📘 *Ejemplo de configuración de Logstash:*

```
input {
  beats {
    port => 5044
  }
}
filter {
  json {
    source => "message"
  }
  mutate {
```

```
    add_field => { "env" => "produccion" }
  }
}
output {
  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "logs-%{+YYYY.MM.dd}"
  }
}
```

Kibana (visualización y análisis)

- Es la **interfaz gráfica** del stack ELK.
- Permite buscar, filtrar y graficar logs con dashboards interactivos.
- Integra vistas para detectar errores frecuentes, analizar tiempos de respuesta y crear alertas.

💡 *Configuración básica en Docker Compose:*

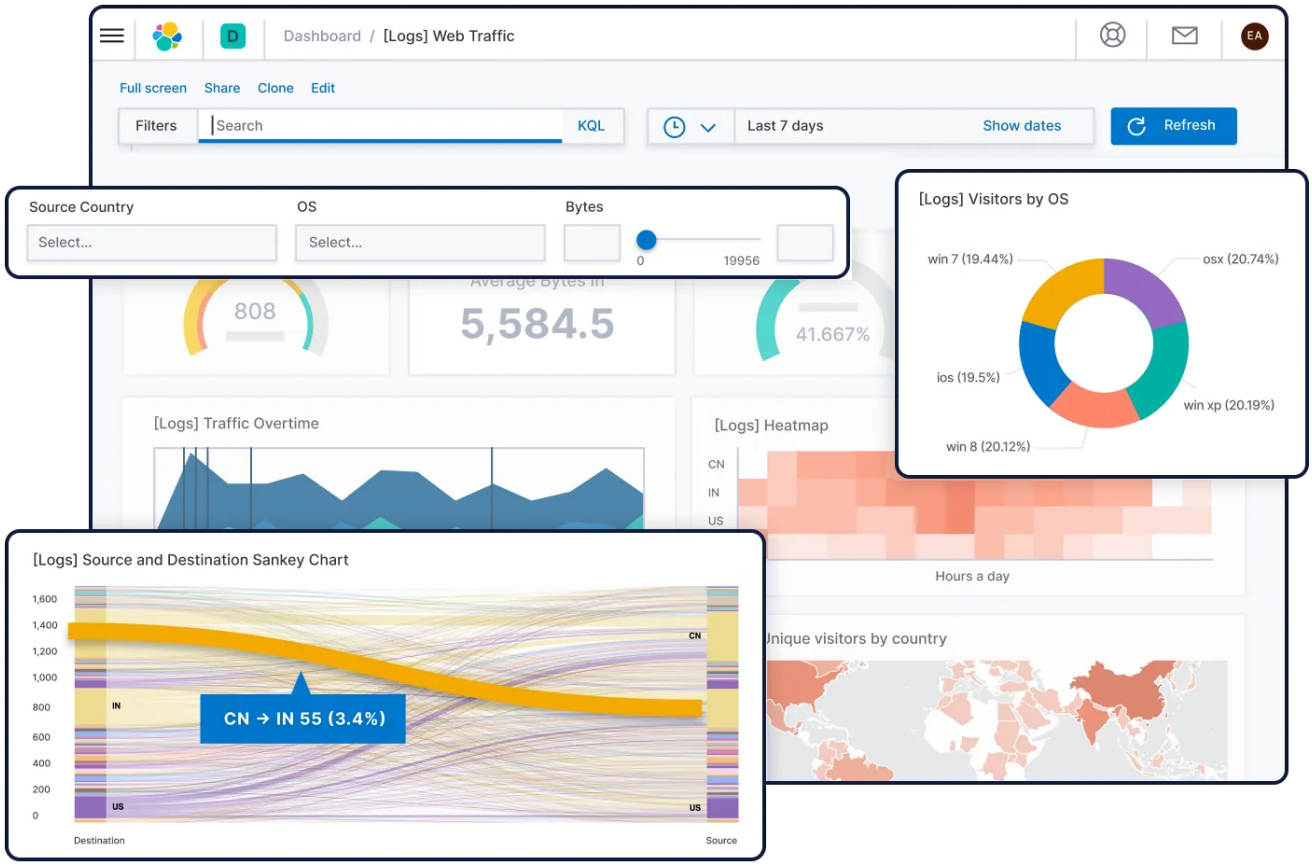
```
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.12.0
    environment:
      - discovery.type=single-node
    ports:
      - "9200:9200"

  logstash:
    image: docker.elastic.co/logstash/logstash:8.12.0
    volumes:
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
    ports:
      - "5044:5044"

  kibana:
    image: docker.elastic.co/kibana/kibana:8.12.0
    ports:
      - "5601:5601"
```

📈 En la interfaz de Kibana, podemos buscar por `correlationId`, agrupar por servicio, o generar dashboards personalizados para errores o tiempos de respuesta.

Para tener una idea visual del resultado que provoca el stack, a continuación agregamos un screenshot de los muchos que se presentan en el sitio del Stack:



> Como se puede observar, **Kibana** es una herramienta con enormes capacidades de configuración para mostrar diferentes vistas del estado general de nuestra solución.
> Su función principal es consumir los datos de los logs almacenados en `Elastic Search` para luego presentarlos a partir de una serie de controles visuales que permiten desde gráficos a líneas de tiempo o incluso algunas herramientas más sofisticadas de análisis de datos.

Loki Stack — Promtail, Loki y Grafana

El **Loki Stack**, creado por **Grafana Labs**, es una alternativa más ligera al ELK Stack. Está optimizado para entornos Docker y Kubernetes, reduciendo el consumo de recursos y la complejidad de configuración.

Loki (almacenamiento e indexación)

- Cumple el mismo rol que Elasticsearch, pero **no indexa el contenido completo** de los logs.
- Solo indexa etiquetas (labels), como el nombre del contenedor, servicio o nivel de log, lo que lo hace mucho más eficiente en almacenamiento.
- Es ideal cuando queremos explorar logs recientes sin requerir búsquedas de texto complejas.

📘 *Ejemplo de consulta en Grafana con Loki:*

```
{service="usuarios-service", level="error"}
|= "Timeout"
```

Promtail (recolección)

- Es el agente que **recoge los logs de los contenedores Docker** (desde `/var/lib/docker/containers`) y los envía a Loki.
- Soporta filtros, etiquetas y envío por red a un servidor Loki centralizado.

📘 *Ejemplo de configuración de Promtail:*

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: docker
    static_configs:
      - targets:
          - localhost
        labels:
          job: docker
          host: demo-host
          __path__: /var/lib/docker/containers/*/*.log
```

Grafana (visualización)

- Proporciona la interfaz de análisis para Loki.
- Permite crear dashboards personalizados y gráficos de logs junto a métricas de Prometheus.
- Soporta filtros por etiquetas, búsqueda textual y correlación temporal.

📘 *Configuración básica en Docker Compose:*

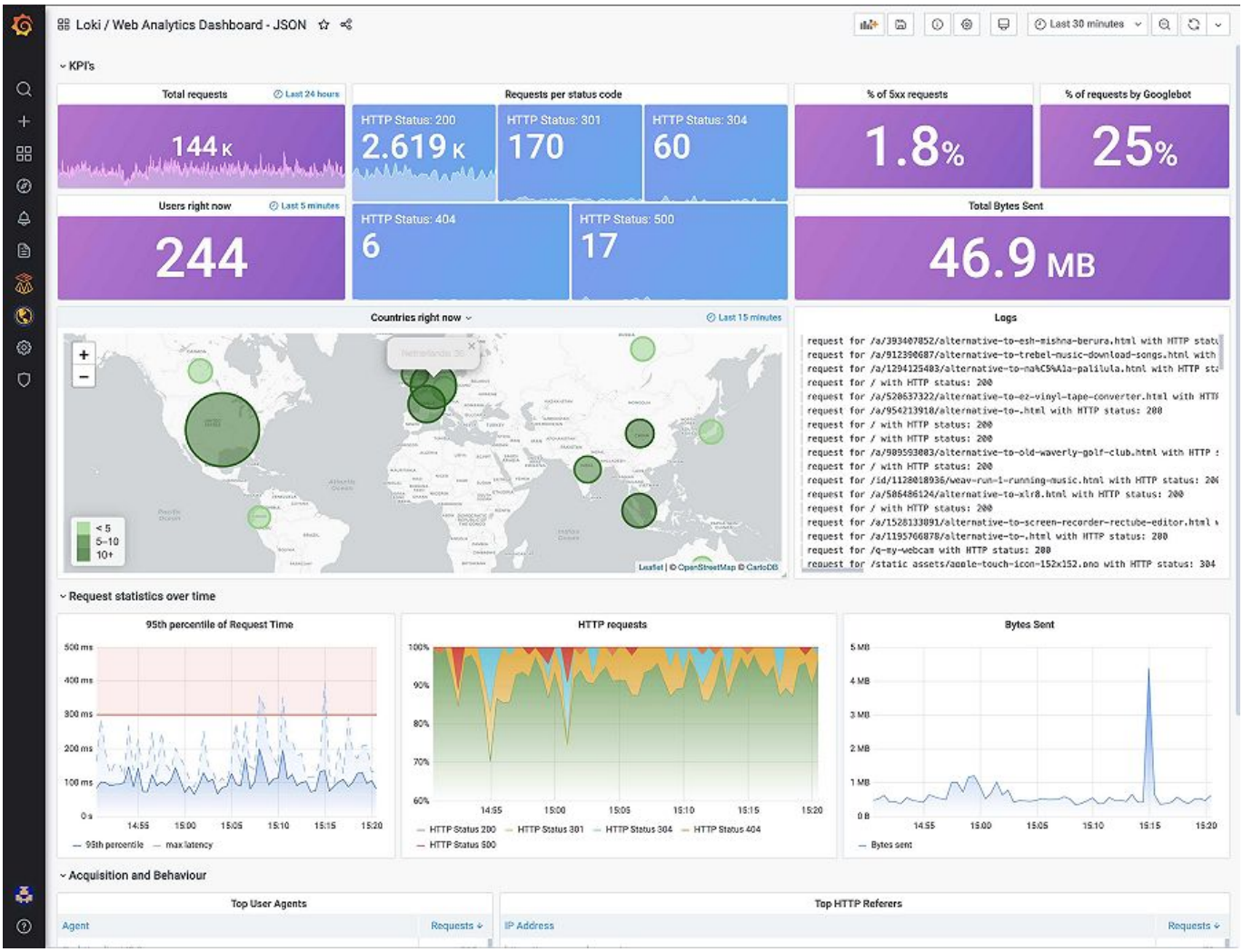
```
services:
  loki:
    image: grafana/loki:latest
    ports:
      - "3100:3100"

  promtail:
    image: grafana/promtail:latest
    volumes:
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - ./config/promtail.yml:/etc/promtail/config.yml

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
```

En la interfaz de Grafana, podemos filtrar fácilmente por etiquetas o por `Correlation ID`, analizar tiempos y visualizar correlaciones con métricas del sistema.

Aquí vamos a agregar también un ejemplo visual del mismo modo que lo hicimos antes con Kibana para tener una idea del resultado:



> Como se puede observar, **Grafana** es otra herramienta con enormes capacidades de configuración para mostrar diferentes vistas del estado general de nuestra solución.
> De un modo similar a `Kibana` se encarga de consumir los datos en este caso desde `Loki` para luego presentarlos a partir de una serie de controles visuales que permiten desde gráficos a líneas de tiempo o incluso algunas herramientas más sofisticadas de análisis de datos.

Comparativa entre ELK y Loki

Característica	ELK Stack	Loki Stack
Complejidad de instalación	Alta (requiere configuración de Logstash y múltiples índices)	Baja (instalación ligera con Promtail + Loki)
Consumo de recursos	Alto (Elasticsearch es intensivo en CPU y memoria)	Bajo (indexa solo metadatos)

Característica	ELK Stack	Loki Stack
Formato de logs	Texto, JSON, estructuras enriquecidas	Texto plano o JSON con etiquetas
Búsqueda	Muy avanzada (filtros, expresiones regulares, agregaciones)	Básica (búsqueda textual y por etiquetas)
Integración con métricas	Limitada (requiere Elastic APM)	Nativa con Grafana y Prometheus
Escalabilidad	Alta, pero costosa	Alta y económica
Ideal para	Grandes volúmenes de logs empresariales	Entornos Docker/Kubernetes, DevOps ágiles

En resumen: ELK vs Loki

- El **ELK Stack** ofrece un análisis profundo y flexible, ideal para entornos corporativos donde se requiere búsqueda avanzada y almacenamiento a largo plazo.
- El **Loki Stack** proporciona una alternativa ligera, rápida y de bajo mantenimiento, perfecta para entornos de microservicios y despliegues Docker.
- Ambos stacks complementan la observabilidad: mientras ELK se enfoca en logs detallados e históricos, Loki facilita la integración con métricas y monitoreo en tiempo real.

> 🎯 En nuestros proyectos, podemos comenzar con Loki por su simplicidad y luego evolucionar hacia un stack ELK cuando el volumen de logs y las necesidades de análisis lo justifiquen.

Conclusiones

- Los logs en entornos Docker deben centralizarse para garantizar trazabilidad y diagnóstico efectivo.
- Actuator permite monitorear el estado y comportamiento de cada microservicio sin herramientas externas.
- La combinación de logs, métricas y trazas constituye la base de la **observabilidad moderna**.
- La integración con ELK o Prometheus/Grafana permite llevar estas prácticas a nivel DevOps profesional.

Bibliografía y recursos recomendados

> 📚 Esta bibliografía complementa los conceptos desarrollados en el *Apunte 24 — Centralización de Logs y Monitoreo de Aplicaciones*, ofreciendo material de lectura técnica, práctica y visual para continuar profundizando en los temas de trazabilidad, observabilidad y DevOps.

Libros y documentación oficial

- **Spring Boot Reference Documentation** — *Spring.io* https://docs.spring.io/spring-boot/docs/current/reference/html/
- **Spring Boot Actuator Guide** — *Spring.io* https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html

- **Micrometer Metrics Documentation** — *Micrometer.io* https://micrometer.io/docs
- **Docker Documentation: Logging Drivers** — *Docker Docs* https://docs.docker.com/config/containers/logging/configure/
- **Grafana Loki Documentation** — *Grafana Labs* https://grafana.com/docs/loki/latest/
- **Elastic Stack (ELK) Documentation** — *Elastic.co* https://www.elastic.co/guide/index.html
- **Logback Project Documentation** — *QOS.ch* http://logback.qos.ch/documentation.html

Artículos y lecturas complementarias

- *"Logging and Observability Patterns in Microservices"* — Martin Fowler https://martinfowler.com/articles/microservice-observability.html
- *"Centralized Logging with ELK Stack"* — DigitalOcean Community https://www.digitalocean.com/community/tutorials/how-to-set-up-the-elk-stack-on-ubuntu-22-04
- *"Introduction to Grafana Loki and Promtail"* — Grafana Labs Blog https://grafana.com/blog/
- *"Building Distributed Tracing in Spring Boot with Sleuth and Zipkin"* — Baeldung https://www.baeldung.com/spring-cloud-sleuth-single-application
- *"Best Practices for Logging in Java"* — LogRocket Blog https://blog.logrocket.com/best-practices-for-logging-in-java/

Videos y recursos multimedia

- 📹 **Spring Boot Actuator Deep Dive** — *Dan Vega* (YouTube) https://www.youtube.com/watch?v=GxvK7bDdO9w
- 📹 **Docker Logging Explained** — *TechWorld with Nana* (YouTube) https://www.youtube.com/watch?v=VfGW0Qiy2I0
- 📹 **Observability with Grafana Loki** — *GrafanaCON* https://www.youtube.com/watch?v=38FfqORIXUo

Repositorios y ejemplos de referencia

- **Spring Boot Actuator Example Project** https://github.com/spring-guides/gs-actuator-service
- **Grafana Loki + Promtail + Docker Compose Example** https://github.com/grafana/loki/tree/main/production/docker
- **ELK Stack with Docker Compose** https://github.com/deviantony/docker-elk